Formalisation and validation of the Std_Logic_1164 and Numeric_Std VHDL packages using the Nqthm theorem prover

J. Dushina, D. Borrione

Formalisation and Validation of the Std_Logic_1164 and Numeric_Std VHDL Packages using the Nqthm Theorem Prover

Julia Dushina Dominique Borrione
Laboratoire TIMA / Université Joseph Fourier
120, rue de la Piscine
UFR IMA, BP 53
38041 Grenoble Cedex 9 FRANCE
E-mail: {Julia.Dushina, Dominique.Borrione}@imag.fr

Abstract

When synthesizing complex digital circuits, it is extremely important to have high confidence in the library of components and functions. This paper deals with the validation of two standardized VHDL packages that were developed for circuit synthesis and which, at the same time, are compatible with enhanced multi-valued simulation tools. The Boyer-Moore theorem prover has been used for this purpose. The paper introduces an efficient and general strategy for utilizing the basic concepts of the Boyer-Moore system to prove the theorems that establish the mathematical properties of the package functions.

1 Introduction

This paper deals with the validation of the Std_Logic_1164 and Numeric_Std packages and is the continuation of earlier work [BBS96] concerned with the Numeric_Bit package. The validation of these packages based on 9-value logic is very important due to their large utilization by the most popular synthesis tools. Designers must trust that the functions of the Std_Logic_1164 and Numeric_Std packages are error-free and consistent. Moreover, the formalization of the internally coherent packages can serve as a basis for future formal verifications of the circuits synthesized from a VHDL description using these packages.

Validation means checking that the packages are internally coherent and that functions as specified have the usually expected mathematical properties. The logic functions defined on the scalar and vector *std_ulogic* types constitute the body of the Std_Logic_1164 package [IEE92]. Numeric_Std defines an arithmetic interpretation for *std_ulogic* vectors, considered to represent signed or unsigned integers; the package includes arithmetic, logic, shift, and comparison functions defined on *std_ulogic* vectors, and some auxiliary functions such as size adjustment [IEE93], [IEE95]. In all function definitions, the size of the vectors and the direction (ascending or descending) of their indices is left unspecified: in the VHDL jargon, all vectors are unconstrained.

To allow for reasoning, the basic VHDL primitives, and the packages constituents have to be formalized in an appropriate formal system. The packages functions being defined for arbitrary vector lengths, their formal verification necessitates general purpose theorem proving techniques. We are using the Nqthm (Boyer and Moore) system for its powerful and largely automated deduction capabilities, and in particular its induction principle.

Previous utilization of Nqthm for the formal verification of hardware can be found in [Hun87], [BHY92], [VVCD92], [Pie91]. A "shallow embedding" of the VHDL language within the Boyer-Moore logic has been performed independently by [Rea94] and [Rus95], where the emphasis was on the formalization of the notion of signals, the process and main statements, and the simulation

algorithm. Less details are provided on the VHDL types, notably the array constructor is abstracted as a list of values as in [Hun87]. In our approach, we represent a vector with all its VHDL semantics, and this paper focuses on vector attributes and 9-value logic. Our work differs from previous ones also in the explicit modelling of error conditions. Contrary to our predecessors, who identify error and base cases in the Boyer-Moore logic, we have introduced an *error* value in our formalization, thus portraying more accurately VHDL statements of the form: "`if erroneous_case then assert false severity error endif`". At the cost of some added proof complexities, we have proven properties of functions for which error cases are specified.

The article is organized as follows. Section 2 introduces briefly the Nqthm theorem prover. Section 3 shows some essential features of the formalization of the VHDL primitives in the Boyer-Moore logic, including the 9-value std_ulogic type. In section 4, we give the basic two level representation of the package functions, and the general stategy for proving properties upon them. The last section outlines the detailed validation process, and summarizes for each group of function the proven properties. We conclude with an assessment of the difficulties we met, and with our perspectives for future work. The `typewriter` style is used throughout the article to identify VHDL or Nqthm code.

2 Boyer-Moore theorem prover

In this section, we outline the basic features of the Boyer-Moore theorem prover to allow easier understanding of further explanation. The interested reader is referred to [BM88] for more details. The basic logic used by the prover is the first order logic without quantifiers and with equality also called the Boyer-Moore or Nqthm logic. The entry language is Lisp-like and relies generally on the notion of a *shell*.

The shell principle allows to introduce a new Nqthm type with the help of the following primitives: constructor, base object, recognizer and accessors (also called destructors). Let us consider how the natural numbers (already defined in the so-called "ground zero logic") are built using this principle. They are introduced by the `add-shell` command depicted in 2.1.

```
(add-shell  add1 zero numberp
          ((sub1 (one-of numberp) zero)) )
```
2.1

```
(add1 (add1 (add1 (add1 (zero)))))      ;the number 4
(numberp 4) = true
(sub1 4) = (add1 (add1 (add1 (zero))))  ;the number 3
(sub1 false) = zero
```
2.2

In this way, the number "4", for instance, is constructed by applying successively four times the constructor `add1` to the initial basic element `zero`. The recognizer `numberp` returns the value `true` if its argument was constructed with the help of `add1` or is the basic element `zero`, and `false` otherwise. The unique accessor `sub1` returns the predecessor of its argument. For example, the predecessor of the number abbreviated as `4` is the number abbreviated as `3` (i.e. `(add1 (add1 (add1 (zero))))`). If the argument of the function `sub1` is not a number i.e. `numberp` returns `false`, then the basic element `zero` will be returned as defined by the last line of the numbers add-shell definition. In fact, the list of accessors defines the arguments that are required by the constructor of a shell. Several examples are depicted below in 2.2.

The shell principle forces every object within the Nqthm logic to be recursively constructed starting with a basic element. A recursive function is then verified to be *well-defined*, i.e. every call to the function terminates. The induction principle is applied for a property demonstration of a well-defined function (the application of a recursive function to the base element constitutes the base step; the applications to an element `X` and `accessor(X)` constitute the induction step of a proof). Only well-defined recursive functions are accepted by the theorem prover.

The introduction of a new type with the help of the add-shell command invokes the automatic creation of a set of axioms associated with this type. The axioms will be used during future theorems proofs. Once a theorem is proved it can be used to prove another theorem.

3 VHDL primitives formalization

Before reasoning about the packages and their functions with the help of Nqthm, it is necessary to formalize initial constructs of VHDL in the Boyer-Moore logic. Our purpose here is not to formalize all the VHDL primitives but only the "relevant and sufficient" ones for the packages verification. This section is devoted to the general vector, VHDL types, and error case formalization.

Notation To ease readability, recognizers are named `p{shell_identifier}`. Moreover, overloaded function names in VHDL are numbered in their Nqthm formalization, to ensure uniqueness of function names. Thus, `f{name_i}` will be used for the i^{th} definition of function "name". The Nqthm comments follow the semicolon sign.

Vector representation In this subsection, we develop a model for bits and general bit-vectors. The formal representation of bit-vectors is central to the task of verifying the functions that operate on them.

The 9 values of the main bit type `std_ulogic` are represented in the Boyer-More logic by literals that are abbreviated with a quote and the corresponding letter.

```
(add-shell  lcons lempty p{list}                                        3.1
            ((lcar (none-of) false)                    ;least significant bit
             (lcdr (one-of p{list} lempty)) )                ;rest of a list

Examples:
(lcons 'X (lcons '1 (lcons '0 (lempty))))                  ;creates list "01X"
(lcar (lcons 'X (lcons '1 (lcons '0 (lempty)))))          ;returns element 'X'
(lcdr (lcons 'X (lcons '1 (lcons '0 (lempty)))))           ;returns list "01"
```

```
(add-shell  vector nil p{vector}                                        3.2
            ((vlist (one-of p{list} lempty)          ;list contained by a vector
             (vleft (one-of numberp) zero)                  ;vector left bound
             (vascending (one-of truep falsep) falsep)) )           ;direction
Example:
(vector (lcons 'X (lcons '1 (lcons '0 (lempty)))) 7 false)   ;vector "01X" (7 downto 5)
```

The representation of a vector should be as close as possible to the initial VHDL form to keep all the attributes of the VHDL vector. On the other hand, the representation has to be simple enough to prove properties of complex functions manipulating bit-vectors. We have chosen to represent a vector at two levels. The lowest one is a list of the values held by a vector: it is similar to the vector notation introduced by Hunt [Hun87]. The command `add-shell` (3.1) is used to create a list type represented by a 2-place constructor `lcons`, a base object `lempty` which is an empty list, a recognizer function `p{list}`, and two destructor functions `lcar` and `lcdr`. The destructor function `lcar` returns the least significant bit of a bit-vector. The destructor function `lcdr` returns the remaining bit-vector with one bit stripped off. If the argument of the `lcdr` function is not a vector, the empty list `lempty` is returned. Some examples are given in 3.1.

The top-level of a vector representation is introduced by the shell `vector` (3.2) and comprises the following elements: a three-place constructor `vector`, a `nil` base object, a recognizer function `p{vector}`, and three destructor functions `vlist`, `vleft`, `vascending` returning the list of the values held by the vector, the vector left bound, and its direction. The vector left bound, direction, and the function returning the list length allow to compute all the other VHDL vector attributes: `'right`, `'length`, `'high`, `'low`, `'ascending`, `'range`, and `'reverse-range`.

Both in package Std_logic_1164 and in Numeric_std, operators and functions that return a vector return it with *normalized* dimensions. However, the normalization adopted in the two packages is different. A vector `VEC` is normalized in Std_logic_1164 if its dimension is `(1 to VEC'length)`; in Numeric_std, it is `(VEC'length-1 down to 0)`. This feature is explicit in our formalization.

VHDL types formalization The syntax of the Nqthm language does not allow to attach its type to a variable. To distinguish between different types a set of recognizer functions has to be introduced. The recognizer of `std_ulogic vector` is defined stepwize, like the bit-vector definition. First, the scalar value is checked to be one of the nine `std_ulogic` values with the help of the `p{std_ulogic}` function. Then the recognizer `p{std_ulogic_list}` controls whether a list contains `std_ulogic` values only. Finally, the recognizer `p{std_ulogic_vector}` first checks its argument to be a vector, and if so, controls a list of the values held by a vector. The `p{std_ulogic}` function models the `std_ulogic` VHDL type and `p{std_ulogic_vector}` function models the `std_ulogic_vector`, `unsigned` and `signed` VHDL types. The "nested" recognizers are depicted in 3.3. The `defn` command followed by a new function name and a list of arguments introduces a new definition. The function `lfix` forces its argument to be an empty list if it is not a list. The function `lemptyp` verifies if its argument is an empty list, and `T` and `F` denote the `TRUE` and `FALSE` values.

3.3
```
;definition of std_ulogic recognizer
(defn p{std_ulogic} (V)                                    ;V is recognized if V is
  (or (equal V 'U) (equal V 'X) (equal V '0) (equal V '1)    ;equal to U or X or...or D
      (equal V 'Z) (equal V 'W) (equal V 'L) (equal V 'H) (equal V 'D) ))

(defn p{std_ulogic_list} (LIST)                ;definition of std_ulogic_list recognizer
      (if (lemptyp (lfix LIST)) T                   ;if LIST is empty, returns True else
          (and (p{std_ulogic} (lcar LIST))         ;returns True if lcar is a std_ulogic
               (p{std_ulogic_list} (lcdr LIST)))) ))      ;and lcdr is a std_ulogic_list

(defn p{std_ulogic_vector} (VEC)             ;definition of std_ulogic_vector recognizer
  (if (p{vector} VEC)                              ;returns True if VEC is a vector and
      (if (p{std_ulogic_list} (vlist VEC)) T F) F))      ;its list is a std_ulogic_list
```

Error case formalization In this subsection, we introduce the formalization of a fundamental notion of the VHDL semantics: the *error*. An error is returned by a functions in the following cases: an actual parameter is of the wrong type, execution failure (e.g. division by zero) or as the result of an *assert* condition returning *false*, with severity *error* or *failure*. Our approach in modeling errors differs from the formalization in HOL proposed by Reetz [Ree95], in that we do not distinguish between elaboration and execution errors, nor between execution errors for functions of different result types.

While formalizing an error notion in the Boyer-Moore logic, the efficiency of proofs and the possibility of error *propagation* in the case of nested function calls have to be considered. For the efficiency, several solutions were explored. I turns out that the error object introduced by the shell-command `(add-shell err nil errorp ())` entails shorter proof times than the error modeled as a function `(defn ERR() 'ErReUr)` [Dus95]. For the propagation, the combination of the `error` with the structure of function definition has to be done as it is shown in the next section. It would be possible to distinguish between different error cases, as it is done by many commercial simulators, by returning varying elements of the shell `err`; in this paper, we consider only one error, in conformance with the LRM and with the packages definition.

4 Formalization and validation general strategy

The couple formalization-validation is tightly binded. The successful formalization of the general notions and functions of the packages in the Nqthm logic entails the effective and elegant theorems proof. Sometimes only certain model allows to demonstrate function properties. In this section, we describe the basic ideas of the packages formalization and validation explaining together the chosen formalization and implied demonstration schema.

Vector functions representation Due to the fact that the vector "core" is a list of values and following the vector definition in the previous section, the packages vector functions are specified in two steps. First, we define a function on lists. Then, we define the same function on vectors, which constructs the *normalized* result vector whose list part is returned by the previously defined function on lists. Thus, the main computation takes place in the list-function body. The example of function and is given in 4.1. Function `land` is a function on lists, defined recursively with the help of `f{and_1}` which is a function on scalar std_ulogic values [Dus95]. The least significant bit is computed first. The vector-level `f{and_2}` function returns a normalized vector (as in the VHDL package), its list part being the result of `land` applied to the list parts of the initial vector arguments.

4.1
```
(defn land (LISTL LISTR)                          ;and on lists
  (if (lemptyp (lfix LISTL)) (lempty)                          ;base case: emtpy list
      (lcons (f{and_1} (lcar LISTL) (lcar LISTR))    ;"and" on least significant bits
             (land (lcdr LISTL) (lcdr LISTR))) ) ))          ;recursive call on lcdr-s

(defn f{and_2} (VECL VECR)                        ;and on vectors
  (if (and (p{std_ulogic_vector} VECL)                       ;parameters type verification
           (p{std_ulogic_vector} VECR))
      (if (equal (vlength VECL) (vlength VECR))              ;length verification: parame-
          (vector (land (VLIST VECL) (VLIST VECR)) 1 T)       ;ters must have equal length
          (err))                                                    ;length error
      (err)) )                                                      ;type error
```

In order to enable the demonstration of functions properties, two kinds of theorems have normally to be proven for every function defined on lists and vectors. These theorems concern the type and size of a function result. In this manner the signature of a function is verified. The theorems related to function `land` are shown in 4.2.

4.2
```
(prove-lemma l_len_land (rewrite)
  (implies (equal (size LISTL) (size LISTR))      ;if parameters are of the same length
           (and (equal (size (land LISTL LISTR))    ;then result is of the same length
                       (size LISTL))                        ;as LISTL parameter and
                (equal (size (land LISTL LISTR))                ;as LISTR parameter
                       (size LISTR))) ))
(prove-lemma l_typ_land (rewrite)
  (implies (and (p{std_ulogic_list} LISTL)          ;if parameters are std_ulogic_list
                (p{std_ulogic_list} LISTR)
                (equal (size LISTL) (size LISTR)))        ;and are of the same length,
           (p{std_ulogic_list} (land LISTL LISTR)) ))  ;then result is std_ulogic_list
```

The chosen vector function representation dictates a common schema for any proof of a function property. First, the property has to be proven on the list-level function. Normally this is the most difficult stage. Then, the proof of the same property on the vector-level function is easily done using the `hint` command of the Nqthm prover. The common strategy of vector functions definition-demonstration is given in figure 1.

Error propagation Using the example of 4.1, we can observe the `err` object application. In the body of the `f{and_2}` function the arguments are first verified to be of the right type. If the arguments do not satisfy the type conditions or are not of the same length, the `err` object is returned. Such a definition corresponds to a real situation during VHDL simulation. Moreover, this construction serves for the `err` object propagation in the case of nested function calls: contrary to simulation which stops computation upon error detection, Nqthm functions, which are total, return an error which is transmitted to the calling function, which in turn has to return an error because its argument is not of the expected type.

Note the general principle of vector-function definition: all argument verifications are made in the vector-level function body, whereas the list-level function is exempted from these extra details, thus simplifying future proofs.

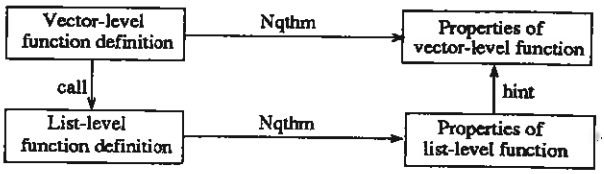

Figure 1: Definition and demonstration strategy

Hierarchical principle for the arithmetic functions This paragraph concerns the arithmetic functions of Numeric_Std, which are much more complex than those of the Std_Logic_1164 package. We shall concentrate on the proof strategy for the functions defined on lists, which is the most difficult. The proof easily extends for the vector functions (see figure 1).

```
(defn add_unsigned_complete (C LISTL LISTR)                                    4.3
    (if (equal (lfix LISTL) (LEMPTY))          ;in the base case (empty parameters)
        (lcons C (lempty))                    ;returns list containing only carry C,
        (lcons                                  ;else constructs a list of a least
          (f{xor_1} C (f{xor_1} (lcar LISTL) (lcar LISTR)))        ;significant bit,
          (add_unsigned_complete               ;and result of add_unsigned_complete
              (f{or_1}                                  ;applied on the calculated
                (f{and_1} (lcar LISTL) (lcar LISTR))                    ;new carry,
                (f{or_1} (f{and_1} (lcar LISTL) C)
                         (f{and_1} (lcar LISTR) C)))
              (lcdr LISTL)                                        ;on (lcdr LISTL) and
              (lcdr LISTR))) ))                                       ;on (lcdr LISTR)

(defn ladd_unsigned (C LISTL LISTR)
  (ltrunc (add_unsigned_complete C LISTL LISTR) (size LISTL) )) ;the most significant
                                                                ;bit is stripped off

(defn l+_A3 (LISTL LISTR)
  (let ((maxsize (f{max} (size LISTL) (size LISTR))))  ;maxsize is the maximum length
  (if (OR (equal LISTL (LEMPTY))                      ;if one of parameters is empty,
          (equal LISTR (LEMPTY)))                                          ;then
      (LEMPTY)                                                ;returns empty list,
      (if (NOT (p_01LH_list LISTL))                      ;else if parameters do not only
          (f_compose LISTL maxsize)                ;contain 0,1,L,H,  f_compose returns
      (if (NOT (p_01LH_list LISTR))             ;list filled with 'X' of length maxsize
          (f_compose LISTR maxsize)                                      ;otherwise
          (ladd_unsigned '0 (f_compose LISTL maxsize)   ;applies ladd_unsigned on the
                            (f_compose LISTR maxsize)) ))) ))    ;modified parameters
```

The functions of Numeric_Std are divided into two groups: internal intermediate functions, and external functions which are exported by the package and can be used by designers. For instance, the VHDL internal add_unsigned function is defined on vectors of the same length and holding values '0' and '1' only. The external '+' function returns the null array if one of its argument is empty, and a vector filled by 'X' if one of its argument contains values other than '0', '1', 'L' or 'H'. Otherwise, it extends the shorter argument to the length of the longest one, converts the 'L' and 'H' bits to the '0' and '1' and finally calls the internal add_unsigned function on the modified arguments.

The formalization and validation of the package functions in the Boyer-Moore logic follows this hierarchical principle, except that one more level is added to ease the proof process. The functions of the added level are the base for the internal and external ones.

Let us consider this approach on the example of the '+' function. We recall that only list-level function formalization is of interest in this paragraph. First, the function add_unsigned_complete

is defined like Hunt's vectors addition [Hunt87]. This is the lowest level function which is called by the function `ladd_unsigned` modeling the internal VHDL function with the same name. At the top of the list functions hierarchy, the function `l+_A3` is defined. These three nested functions are depicted in 4.3. The use of the function `ltrunc` allows to strip off the most significant bit of the `add_unsigned_complete` result, thus returning a list of the same length as the length of its arguments. The function `f_compose` normalizes the arguments (size adjustment and forcing the 'L' and 'H' values) or returns the list filled by 'X'. The subsequent definition, in the Nqthm logic, of '+' on vectors is straightforward.

When proving properties, the hierarchical principle is kept: the theorems of the lower level are used during the demonstrations of the higher levels as shown in figure 2. In this way, the desired properties of the external functions are easily and quickly proven, whereas it is very difficult and sometimes impossible to demonstrate them directly.

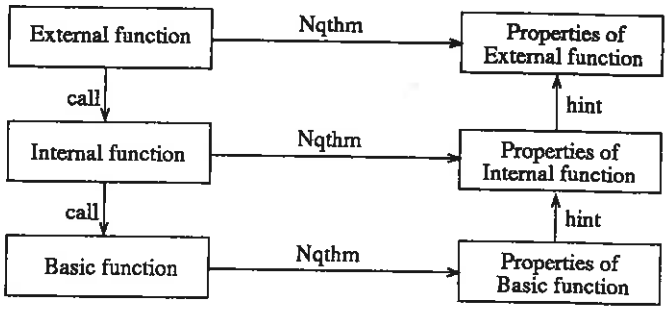

Figure 2: Three levels of the Numeric_Std functions

Functions validation The validation consists of checking that the functions perform as intended. To do so, mathematical properties of functions must be proven. Mathematical properties can include, according to the concrete function, several features:

- commutativity
- associativity
- type and size of result (the signature of a function)
- arithmetic interpretation of a function

Obviously, the theorem concerning the size of a result can be proven only for the functions defined on lists and vectors.

The *arithmetic interpretation* is the essential mathematical property to verify. It links a function defined on vectors to its meaning from the "natural number" point of view.

In the VHDL package, the function $to\_integer(V)$ produces the natural number binary-encoded by the bit-vector V. Conversely, the function $to\_unsigned\ (N, size)$ produces the bit-vector which is the binary code of $(N\ mod\ 2^{size})$ number. The validation of these functions in Nqthm consists in the proof of a double conversion theorem which states that if a bit-vector is converted into a natural number and then back into a bit-vector of the same length, the initial and final vectors are the same.

The arithmetic interpretation of the result of each function defined on bit vectors defines the function semantics. Thus, for each external operator "op" of the package, we prove a theorem which relates the operator on bit_vectors to its corresponding operator on naturals, which takes the general template:

$$to\_integer(a\ \ op\ \ b) = (to\_integer(a)\ \ op\ \ to\_integer(b))$$

For example, the natural number represented by (add_unsigned_complete c a b) is the arithmetic sum of those represented by a, b, and c. For the functions add_unsigned and 1_A3 (4.3) the arithmetic interpretation is expressed by the following equation:

$$to\_integer(a + b) = (to\_integer(a) + to\_integer(b)) \ \ mod \ \ 2^{maxsize\ (a,\ b)}$$

The presence of $mod\ 2^{maxsize\ (a,\ b)}$ is due to the fact that "+" as defined in the package cuts the output carry. The demonstration of arithmetic interpretations, as all complex theorems, requires a number of auxiliary lemmas.

5 VHDL synthesis packages validation

This section comprises the specific functions and associated properties formalized and demonstrated during the packages validation.

5.1 Std_Logic_1164 package

For the basic Std_Logic_1164 package the logic functions AND, OR, XOR, NAND, NOR, XNOR, and NOT were defined on scalar std_ulogic values and vector of std_ulogic by means of auxiliary functions defined on std_ulogic lists. For both groups the main properties of logical functions were proved: commutativity, associativity, theorems of De Morgan, and the theorem called negation of negation (NOT(NOT X)=X). It has to be noted that all these theorems for the functions defined on the VHDL types (std_ulogic scalar values and vectors) were proven in their most general form. i.e. without preconditions. This is very important for their future utilization: the more general a theorem, the more applicable it is. The associativity theorem for the function and defined on vectors is depicted in 5.1.

5.1
```
(prove-lemma l_ass{and_2} (rewrite)
   (equal (f{and_2} (f{and_2} VEC1 VEC2) VEC3)          ;((VEC1 and VEC2) and VEC3)
          (f{and_2} VEC1 (f{and_2} VEC2 VEC3)) ))       ;(VEC1 and (VEC2 and VEC3))
```

5.2
```
(prove-lemma l_cvt{and_2} (rewrite)
  (implies (and (p{std_ulogic_vector} VECL)                ;if VECL and VECR are
                (p{std_ulogic_vector} VECR)                 ;std_ulogic_vectors
                (equal (vlength VECL) (vlength VECR)))    ;and are of the same length,
           (equal (f{to_ux01_vector} (f{and_2} VECL VECR))   ;then converted result of
                  (f{and_2} (f{to_ux01_vector} VECL)     ;"and" function is the same as
                            (f{to_ux01_vector} VECR))))) ;"and" of converted parameters
```

Other package functions such as the conversion from the main std_ulogic type to one of its subtypes were also defined. A subtype is a subset of the initial nine logic values. The proved properties are based on the fact that logical functions always return a value of the UX01 subtype. In this way, we have demonstrated that the result of a logical function converted to this subtype is equal to the result of the same logical function applied to the arguments converted to the UX01 subtype. Such a theorem for the and vector-function is outlined in 5.2. The body of the f{to_ux01_vector} conversion function is not shown here.

5.2 Numeric_Std package

Numeric_Std is a big package that includes not only principal arithmetic functions but also a set of auxiliary and minor functions used by the principal ones. The package functions formalization and validation requires the hierarchical approach that was shown on the example of the addition. This subsection discusses the main function groups of the Numeric_Std package. Only the list-level functions are considered here, for more detail see [Dus95].

Subtraction function. The VHDL external subtraction function calls directly the internal addition function **add_unsigned**. We have slightly modified this schema. First, the additional function **sub_unsigned** depicted in 5.3 has been defined. Then, the external subtraction function calls **sub_unsigned** applying it on the arguments modified by the **f_compose** function.

```
(defn sub_unsigned (LISTL LISTR)                                    5.3
  (ladd_unsigned  '1 LISTL (lnot LISTR)) )
```

The validation consists of the function signature (type and size of the result) and arithmetic interpretation theorems. The arithmetic interpretation of (**sub_unsigned a b**) function (a-b) is splitted into two cases. In the first case, the natural number represented by the a-vector is greater than the number represented by the b-vector: the natural number of (**sub_unsigned a b**) is the mathematical subtraction of the numbers represented by a and b. The second case concerns the situation when the natural interpretation of a is less that of b. The arithmetic interpretation is then expressed by the following equation:

$$to\_integer(a-b) = 2^{size\ of\ result} - (to\_integer(b) - to\_integer(a)).$$

All these theorems were demonstrated for the top-level subtraction function (**l-_A9 a b**) under the assumption that a and b are the lists containing 'L', 'H', '0' and '1' values only.

Multiplication The basic idea of the multiplication formalization is a special shift-left function, that increments the size of the initial list by the number of shifted bits. The multiplication and shift-left functions are shown in 5.4.

```
(defn lxsll-not_trunc (LIST N)                                                    5.4
  (lappend (lall-zeros N) LIST))          ;example: lxsll-not_trunc ("01X" 3) = "01X000"

(defn tmp_mult (RES LISTL LISTR)        ;(LISTL * LISTR) + RES
  (if (lemptyp (lfix LISTL))   RES           ;if parameters are empty, returns RES, else
      (if (equal (lcar LISTL) '0)           ;if the least significant bit of LISTL is 0,
                 (tmp_mult (lresize_R2 RES (add1 (size RES)))   ;then shifts left LISTR
                           (lcdr LISTL)                          ;by one position, and
                           (lxsll-not_trunc LISTR 1))      ;considers next bit of LISTL,
                 (tmp_mult (add_unsigned_complete '0 RES LISTR) ;else adds LISTR to RES,
                           (lcdr LISTL)                          ;shifts LISTR, and
                           (lxsll-not_trunc LISTR 1)) )))  ;considers next bit of LISTL
```

The function lappend appends the N least significant '0' to the list LIST. The algorithm of the auxiliary function **tmp_mult** is usual: according to the least significant bit of the argument LISTL, the shifted by one bit argument LISTR is either added to the result of the previous iteration or just transmitted to the next call of **tmp_mult**. It is assumed that **tmp_mult** will be called with the argument RES equal to zero (i.e. with the list containing the '0' values only). The size of the **add_unsigned_complete** result is one bit more than the size of its arguments. At the end, the size of the resulting list is the sum of the LISTL and LISTR sizes. The top-level multiplication function first forces its arguments to contain the '0' and '1' values only and then applies the **tmp_mult** function.

In order to validate the multiplication, the function signature and arithmetic interpretation were proven. The arithmetic interpretation consists of proving that the natural representation of the resulting list is really the multiplication of the natural representations of the arguments. The proof is done in two steps. First, the theorem is demonstrated in general form, i.e. for any valuable list of RES (not only for zero-list). The natural result is, in this case, the multiplication of the natural counterparts of LISTL and LISTR plus the natural number corresponding to RES. Second, the theorem is proved for RES containing the '0' values only.

Shift and Rotate functions These functions are defined in the Boyer-Moore logic with the help of auxiliary functions like `ltrunc` and `lappend` and semantically correspond to the VHDL definitions. The example of the internal shift-left function is shown in 5.5. The arithmetic interpretation theorem of this function is based on the fact that a left shift by one position multiplies the initial natural number by 2.

```
(defn lxsll (LIST N)                                                    5.5
  (if (LEQ N (size LIST))            ;if number N of shift position <= length of LIST,
      (lappend (lall-zeros N)                              ;then shifts left LIST by N
               (ltrunc LIST (difference (size LIST) N)))                ;and pads with 0
      (lall-zeros (size LIST)) ))                        ;else returns list filled by 0
                                    ;the result list is of the same length as LIST
```

For the shift-right group the arithmetic interpretation asserts that every right shift by one bit divides the initial natural number by two. Only the function signature theorem can be proven for the rotate functions series.

Comparison functions It is of special interest to see how the comparison functions were represented in the Boyer-Moore logic. Due to the chosen list representation, the only way to process a list is to start from the least significant bit. Contrary to arithmetic operators, for relational functions the reverse order is wanted. We show in 5.6 the example of the `lunsigned_less` function. The function recursively calls itself till the arguments contain only one (the most significant) bit, making comparison obvious. If at some iteration `LISTL` is still equal to `LISTR`, then the current least significant bits are compared. With this definition it was possible to demonstrate that the `lunsigned_less` relation on lists corresponds to the `<` operator on their natural number interpretation.

```
(defn lunsigned_less (LISTL LISTR)                                      5.6
  (if (lemptyp (lfix LISTL))                        ;in the base case (empty parameters)
      F                                                          ;returns False;
      (if (lemptyp (lcdr LISTL))                ;if parameters contain only one element,
          (LESSP (lcar LISTL) (lcar LISTR))                ;then compares these elements,
          (if (lunsigned_less (lcdr LISTL) (lcdr LISTR))       ;else compares recursivly
              T                        ;lcdr-s: returns True if (lcdr LISTL)<(lcdr LISTR);
              (if (lunsigned_equal (lcdr LISTL) (lcdr LISTR))   ;if they are equal, then
                  (LESSP (lcar LISTL) (lcar LISTR)  ;compares the least significant bits
                  F)))) )                   ;if (lcdr LISTL)>(lcdr LISTR), returns False
```

6 Conclusion

In this paper the formal validation of two VHDL synthesis packages with the help of the Boyer-Moore Nqthm theorem prover was presented. The designers can have more confidence in their algebraic properties while using them during the synthesis process.

During the formalisation and validation process, 163 functions were defined and 461 theorems were demonstrated, three quarters of which are auxiliary. The table below summarizes the validation process: the functions and their properties in the upper rows of the table are used by the functions of the lower rows. This hierarchical approach, illustrated in this paper on the example of the addition function, was systematically applied.

The main result of this work is the elaboration of a method for the automatic generation of a library of functions and theorems in the input language of Nqthm, from VHDL package definitions. Based on this study, a compiler has been implemented, that automatically produces the recognizer functions for enumerated data types, and function definitions on vectors of any type, together with the automatic generation of the appropriate "function signature" theorems. This constitutes a first

STD_LOGIC_1164 PACKAGE

Functions	*Proven properties*
Base definitions of list and vector	Theorems about the size and type of general list and vector
Scalar group	
Type recognizers and Logic functions	Commutativity, Associativity, Theorems de Morgan, Negation of negation
Convertion and resolution functions	Theorem about the type of the result
List group	
Type recognizers and Logic functions	Signature, Commutativity, Associativity, Theorems de Morgan, Negation of negation
Convertion and resolution functions	Theorems about the type of the result
Vector group	
Type recognizers and Logic functions	Signature, Commutativity, Associativity, Theorems de Morgan, Negation of negation
Convertion and resolution functions	Theorems about the type of the result

NUMERIC_STD PACKAGE

Functions	*Proven properties*
List group	
Auxiliary functions like "cut or append list" (defined by ourself)	Signature, Arithmetic interpretation, Auxiliary theorems
Internal auxiliary functions of the package	Signature, Double convertion, Arithmetic interpretation
Addition defined by ourself	Signature, Commutativity, Arithmetic interpretation
Internal addition function	Signature, Commutativity, Arithmetic interpretation
External addition function	Signature, Commutativity, Arithmetic interpretation
Subtraction defined by ourself	Signature, Arithmetic interpretation
External Subtraction function	Signature, Arithmetic interpretation
Multiplication defined by ourself	Signature, Arithmetic interpretation
External Multiplication function	Signature, Arithmetic interpretation
Internal Shift and Rotate functions	Signature, Arithmetic interpretation
External Shift and Rotate functions	Signature, Arithmetic interpretation
Internal Comparison functions	Signature, Relation preserved on natural numbers
External Comparison functions	Signature, Relation preserved on natural numbers
Vector group	
Type recognizers	Theorem implying the type of the list contained by a vector from the type of the vector
Internal auxiliary functions of the package	Signature, Double convertion, Arithmetic interpretation
Internal addition function	Signature, Commutativity, Arithmetic interpretation
External addition function	Signature, Commutativity, Arithmetic interpretation
External subtraction function	Signature, Arithmetic interpretation
External multiplication function	Signature, Arithmetic interpretation
Internal Shift and Rotate functions	Signature, Arithmetic interpretation
External Shift and Rotate functions	Signature, Arithmetic interpretation
Internal Comparison functions	Signature, Relation preserved on natural numbers
External Comparison functions	Signature, Relation preserved on natural numbers

step towards the fully automatic verification of behavioral VHDL specifications, using theorem proving techniques.

Future works include the investigation of the use of the more recent ACL2 theorem prover, which provides a unified built in formalization of the relative integers (naturals and negatives are different shells in Nqthm, and a shell for integers had to be defined together with all operators and their properties). We expect from ACL2 a significant performance improvement for the validation of all the functions on `SIGNED` bit vectors, while retaining the fully automatic proof capabilities that motivated our initial selection of Nqthm.

References

[BBS96] Borrione Dominique, Bouamama Hakim, and Suescun Rodolphe. Validation of the Numeric_Bit package using NQTHM theorem prover. In *Proc. of the Third Asia Pacific Conference on Hardware Description Languages (APCHDL'96)*, pages 168–172, Bangalore, India, January 1996. IFIP WG 10.5.

[BHY92] Brock B.C., Hunt W.A., and Young W.D. Introduction to a formally defined hardware description language. In *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 3–35, Nijmegen, June 1992. IFIP TC10/WG 10.2, North-Holland.

[BM88] Boyer R.S. and Moore J S. *A Computational Logic Handbook.* Academic Press, 1988.

[Dus95] Dushina Julia. Formalisation de la logique à neuf valeurs et validation de paquetages VHDL pour la synthèse. Master's thesis, Institut National Polytechnique de Grenoble, ENSIMAG, September 1995.

[Hun87] Hunt W.A. The Mechanical Verification of a Microprocessor Design. In D.Borrione, editor, *From HDL Description to Guaranteed Correct Circuit Designs*, pages 89–129. Elsevier Science Publishers B.V., 1987.

[IEE92] IEEE VHDL Synthesis Working Group (IEEE 1076.3). *IEEE Std_Logic_1164 logic package.* ftp::/vhdl.org/vi/libutil/utilities/gen_functions/IEEE1164, 1992.

[IEE93] IEEE Standard 1076-1993. *Standard VHDL Language Reference Manual*, 1993.

[IEE95] IEEE VHDL Synthesis Working Group. *Draft Standard VHDL Synthesis Package 1076.* WWW access://vhdl.org/vi/vhdlsynth/vhdlsynth.html, April 1995.

[Pie91] Pierre Laurence. One Aspect of Mechanizing Formal Proof of Hardware: the Generalization of Partial Specifications. In *Proc. ACM International Workshop on Formal Methods in VLSI Design*, Miami, January 1991.

[Rea94] Read Simon. *Formal Methods for VLSI Design.* PhD thesis, The University of Manchester, 1994.

[Ree95] Reetz Ralf. Deep Embedding VHDL. In *International Workshop on Higher Order Logic Theorem Proving and its Applications*, Aspen Grove, Utah, USA, September 1995.

[Rus95] Russinoff David M. A Formalization of a Subset of VHDL in the Boyer-Moore Logic. In *Formal Methods in System Design*, volume 7, pages 7–25, August 1995.

[VVCD92] Verkest D., Vandenbergh J., Claesen L., and De Man H. A Description Methodology for Parameterized Modules in the Boyer-Moore Logic. In Stavridou V., Melham T.F., and Boute R.T., editors, *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 37–57, Nijmegen, June 1992. IFIP TC10/WG 10.2, North-Holland.